Probabilistic Dataflow Analysis using Path Profiles on Structure Graphs

Arun Ramamurthi
Microsoft India R & D Pvt. Ltd.
Hyderabad, India
aramamu@microsoft.com

Subhajit Roy
Indian Institute of Technology,
Kanpur, India
subhajit@iitk.ac.in

Y. N. Srikant
Indian Institute of Science,
Bangalore, India
srikant@csa.iisc.ernet.in

ABSTRACT

Speculative optimizations are increasingly becoming popular for improving program performance by allowing transformations that benefit frequently traversed program paths. Such optimizations are based on dataflow facts which are mostly true, though not always safe. Probabilistic dataflow analysis frameworks infer such facts about a program, while also providing the probability with which a fact is likely to be true. We propose a new Probabilistic Dataflow Analysis Framework which uses path profiles and information about the nesting structure of loops to obtain improved probabilities of dataflow facts.

Category and Subject Descriptors D.3.4 [*Processors*]: Compilers, optimization

General Terms Algorithms

1. INTRODUCTION

Control-flow profiling gives us information about which nodes, edges or program paths are executed frequently during program execution. Previous attempts at using profiling information for probabilistic dataflow analysis have used edge or two-edge profiling techniques [5][9]. We have used *path profiles* [1][4] which provides more details than edge profiles because of factors like branch correlation.

We have merged the ideas of context tupling [3], use of path profiles [1][7] and structure graphs [4] to develop a new probabilistic data flow analysis framework. A probabilistic dataflow solution is similar to a normal dataflow analysis solution except that it is annotated by dataflow facts with probabilities. Such solutions can be used for improving performance by techniques like speculative data prefetching where our probability values can be used to prune out less-likely dataflow facts below a threshold value.

Related papers: Ammons et al.[1], Mehofer et al.[9], Silva et al.[5] are publications similar to our work (compared in Related Work section).

2. BACKGROUND

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.
Copyright 2011 ACM 978-1-4503-0443-6/11/09 ...$10.00.

2.1 Structure Graphs

The key idea of Structural Path Profiling [4] is to profile each loop independently. So, the program can be seen as a hierarchy of such nested structure graphs. Hierarchical Path Profiling [12] also proposes similar ideas.

We define a *summary node* as the representative header node of a nested loop. An *exit node* is the target of a loop-exit edge. All other nodes are called *regular nodes*.

Example: Figure 2(b) shows two summary nodes - c, g which get expanded in the inner structure graphs, Figure 2(c), 2(d) respectively. The inner structure graphs include regular nodes, each representing a basic block of the inner loop. A structure graph also contains various exit nodes of the loop such as g, j in Figure 2(c). For the outline structure graph in Figure 2(a), the end node (j) is the exit node.

A *loop-path* is a path which begins at the loop header and terminates at the source of the back edge. A *loop-exit path* is a path which begins at a loop header and terminates outside the loop.

2.2 Path Profiles

Ammons et al.[1] used acyclic path profiles [7] to improve dataflow solutions by separating facts propagating along frequent paths from the others. Their algorithm constructs a Labeled Transition System(LTS) (as briefed below) to track the propagation of dataflow facts along the hot paths:-

- q_i , a node in the LTS corresponds to a state
- $(q_1 \rightarrow^{k_1} q_2 \rightarrow^{k_2} q_3 \rightarrow \ldots \rightarrow^{k_{i-1}} q_i)$ is a path in the LTS if there exists a hot path $k_1 \rightarrow k_2 \rightarrow \ldots \rightarrow k_{i-1}$ in the CFG.
- The CFG nodes transform into edge labels on the LTS.

For our running example, the LTSs for the structure graphs in Figure 2 are shown in Figure 3. **Example:** In Figure 2(c) the path $c \rightarrow d \rightarrow f \rightarrow g$ transforms to the path $1 \rightarrow^c 2 \rightarrow^d 3 \rightarrow^f 4 \rightarrow^g 5$ in Figure 3(c).

3. APPLYING PATH PROFILE INFORMATION ON STRUCTURE GRAPHS

3.1 Probability definitions for structure graphs

Our probabilistic dataflow analysis (like any other) computes the probability of a dataflow fact at every program point. We use an *Structural path profiler* to get statistics regarding frequencies of acyclic paths in a function. We then use these frequencies to estimate the probabilities of facts in accordance with the profiler by focusing on each natural loop (its structure graph) separately.

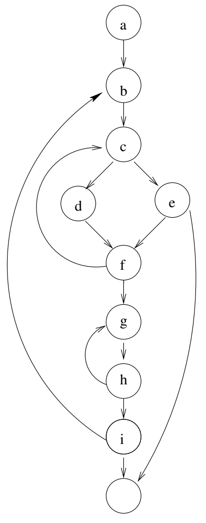

Figure 1: Our running example: CFG with 3 loops.

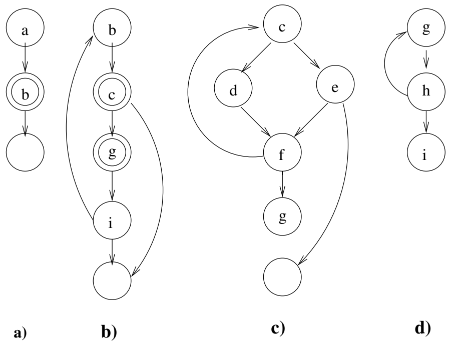

Figure 2: The four structure graphs for the CFG in Figure 1.

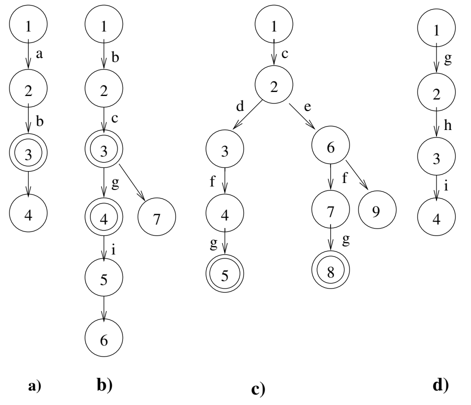

Figure 3: The labeled transition diagrams for structure graphs in Figure 2.

For a CFG, let there be a node n with loop nest level of k and the header nodes be $n_1, n_2, .., n_k$ for each of the corresponding structure graphs $L_1, L_2, .., L_k$ respectively. Node n_j will also be a summary node in the structure graph L_{j-1}.

- For any path p_i starting from header node n_j and ending at node n_{j+1} in structure graph L_j, $Prob(p_i)$ w.r.t L_j = (frequency of path p_i) / (sum of frequencies of all paths reaching n_{j+1}) where n_{j+1} is the summary node for the next nested structure graph. Frequency of path p_i is the number of times path p_i is traversed during program execution as seen by the profiler.
- The probability of reaching node n_{j+1} from the header node n_j of structure graph L_j is the sum of probabilities of reaching n_{j+1} via all possible paths from n_j to n_{j+1} in L_j.
- The probability of reaching node n from the start node of the CFG is the product of probabilities of reaching node n_{j+1} from the header node n_j in L_j for all structure graphs, 1 to k. For the last k^{th} structure graph, n_{j+1} will be node n. The probabilities across different loops are multiplied since all the acyclic paths are in different loops, and hence we assume them to be independent for simplicity.

Example: The probability of reaching h (Figure 1) in the CFG is equal to the product of probability of reaching b (from a) in Figure 2(a), probability of reaching g (from b) in Figure 2(b) and probability of reaching h (from g) in Figure 2(d).

3.2 Labeled Transition Systems (LTS)

An LTS is a deterministic finite automaton that recognizes hot paths. Our analysis creates a separate LTS for each structure graph of the CFG. The assignment of probabilities on the different LTSs is done by handling loop-paths and loop-exit paths separately. We argue that merging the loop path and loop exit path information (as in the case of profiling the entire CFG together) leads to assigning a small probability to a loop-exit node which is incorrect as that node would be accessed with a high probability irrespective of the number of loop iterations.

For Figure 3(c), let us assume the frequencies of loop-paths $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, $1 \rightarrow 2 \rightarrow 6 \rightarrow 7$ are f_1 and f_2 respectively. Similarly the frequencies of loop-exit paths $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$, $1 \rightarrow 2 \rightarrow 6 \rightarrow 7 \rightarrow 8$ and $1 \rightarrow 2 \rightarrow 6 \rightarrow 9$ are f_3, f_4 and f_5 respectively.

The probability of a loop path relative to the loop-structure graph is the frequency of that path divided by the sum of frequencies of all loop paths for that loop in the LTS. In Figure 3(c), the probability of the reaching node f along path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ is $\frac{f_1}{f_1+f_2}$. Similarly, the probability of a loop-exit path is calculated w.r.t all loop-exit paths for that loop. The probability for the loop in Figure 2(c) to exit via node j is $\frac{f_5}{f_3+f_4+f_5}$.

The computed probability of each path is assigned to its last state in the LTS and thereafter propagated bottom-up. At each state in the LTS, the probability would be equal to the sum of probabilities of its loop node successors. However, we do not propagate loop-exit probabilities.

The probability of reaching a node in the structure graph would be equal to the sum of the probabilities of all paths by which the node could be reached in the LTS. So, the probability of reaching node f w.r.t loop in Figure 2(c) is computed as $Prob(1 \rightarrow 2 \rightarrow 3 \rightarrow 4) + Prob(1 \rightarrow 2 \rightarrow 6 \rightarrow 7)$ in the LTS in Figure 3(c).

The actual probability of reaching a node in the CFG would be equal to the sum of all path probabilities by which that node could be reached. So, the overall probability of reaching node f is computed as $Prob(a \rightarrow b \rightarrow c \rightarrow d \rightarrow f) + Prob(a \rightarrow b \rightarrow c \rightarrow e \rightarrow f)$.

This path probability would be computed by just multiplying state probabilities in the LTS. To compute path probability from the start of the CFG, we need to multiply path probabilities across different LTS. So, the total probability of reaching node f via path $a \rightarrow b \rightarrow c \rightarrow d \rightarrow f$ is the product of $Prob$(reaching 3 in Figure 3(a)), $Prob$(reaching 3 in Figure 3(b))$andProb$(reaching 4 in Figure 3(c)).

In Figure 2(d), there is one loop path $g \rightarrow h$ and one loop exit path $g \rightarrow h \rightarrow i$. So, we assign a probability of 1.0 to node 4 in Figure 3(d) which means that a dataflow fact generated at f and doesn't get killed in g or h should still have a probability of 1.0 at i.

4. OUR PROBABILISTIC DATAFLOW ANALYSIS ALGORITHM

Ammons et al.[1] suggested the idea of using Context Tupling [3] for dataflow analysis - where a state (in the LTS) is a context associated with a node in the CFG. We use this approach and map each context into a probability value for the tupled dataflow fact. For our approach, we create a separate LTS for each structure graph.

4.1 Contexts and context strings

We define a *context* as a state in a given LTS. It is represented by the tuple, $\langle LTSid, state \rangle$ where each LTS is identified by a unique $LTSid$. We define a *context string* as a string of contexts where each context represents a state in a different LTS. A context string uniquely identifies a path from the start node in the CFG.

While traversing a path we switch across different LTSs (or loops). One of the contexts in the context string is the "active" context: this context is used to trace facts in the current LTS; while the remaining string is the "latent" context string. We show a context as "q.c", where "q" is latent (corresponding to other LTSs) in the context string and "c" is the active context. Table 1 shows the computation of context strings after the first iteration on the nodes of the CFG where A, B, C, D refer to the LTSs in Figure 3 respectively. For node e, $(A, 3), (B, 3)$ is the latent string while $(C, 6)$ is the active context. Node e's active context pertains to the LTS in Figure 3(c) since the innermost loop to which e belongs is the one in Figure 2(c)).

We associate a context string with each dataflow fact. We propagate a dataflow fact to its successor *only if there is a valid transition from the active context in the context string on the label of the successor.* For node e, it's predecessor is c with an active context of $(C, 2)$. We look for a transition from node 2 on the label e and find node 6 in the LTS in Figure 3(c). Thus, the active context for node e is $(C, 6)$ with the latent context string remaining unchanged. Note that node f has two context strings for two different paths $a \to b \to c \to d \to f$ and $a \to b \to c \to e \to f$. Similarly j has three context strings, the third one pertaining to the path $a \to b \to c \to e \to j$.

Entering and leaving an LTS

We enter another LTS if we encounter a summary node n in the current LTS. We append the relevant context of the inner loop LTS to the current context string. Also, as we are switching across LTSs (entering the nested LTS), we change the active context to that of the nested LTS in the following manner: we change the context string, $d^{q.a} \to d^{(q+a).b}$ where d is the dataflow fact, q is the latent context and a was the active context before entering the nested LTS. $q+a$ appends a to the latent string q. The active context b is the state of the node n in the inner loop LTS which is now the active LTS.

Example: At node c, the context string gets updated from $\left\langle (A, 3), \underline{(B, 3)} \right\rangle$ to $\left\langle (A, 3), (B, 3), \underline{(C, 2)} \right\rangle$. $(B, 3)$ is a summary node due to which we enter the header node 1 in the inner loop LTS (Figure 3(c)). Here, there is a transition from 1 on the label c to node 2.

Table 1: The context strings for the nodes of the CFG in Figure 1 after the first iteration of dataflow analysis. (The underlined context is the active context.)

Node	Context string
a	$\left\langle \underline{(A,2)} \right\rangle$
b	$\left\langle (A,3), \underline{(B,2)} \right\rangle$
c	$\left\langle (A,3), (B,3), \underline{(C,2)} \right\rangle$
d	$\left\langle (A,3), (B,3), \underline{(C,3)} \right\rangle$
e	$\left\langle (A,3), (B,3), \underline{(C,6)} \right\rangle$
f	$\left\langle (A,3), (B,4), \underline{(C,4)} \right\rangle, \left\langle (A,3), (B,4), \underline{(C,7)} \right\rangle$
g	$\left\langle (A,3), (B,4), (C,5), \underline{(D,2)} \right\rangle, \left\langle (A,3), (B,4), (C,8), \underline{(D,2)} \right\rangle$
h	$\left\langle (A,3), (B,4), (C,5), \underline{(D,3)} \right\rangle, \left\langle (A,3), (B,4), (C,8), \underline{(D,3)} \right\rangle$
i	$\left\langle (A,3), \underline{(B,5)}, (C,5), (D,4) \right\rangle, \left\langle (A,3), \underline{(B,5)}, (C,8), (D,4) \right\rangle$
j	$\left\langle \underline{(A,4)}, (B,6), (C,5), (D,4) \right\rangle, \left\langle \underline{(A,4)}, (B,6), (C,8), (D,4) \right\rangle$ $\left\langle \underline{(A,4)}, (B,7), (C,9) \right\rangle$

We leave an LTS if we encounter an exit node n in the current LTS. In this case, we switch back to the previously active LTS (that of the outer loop) and make that as the active LTS. However, we retain the context of the inner LTS which has now been exited. The formal definitions are similar to the one while entering an LTS and have been skipped.

Example: One context string at node h is updated from $\left\langle (A, 3), (B, 4), (C, 5), \underline{(D, 3)} \right\rangle$ to $\left\langle (A, 3), \underline{(B, 5)}, (C, 5), (D, 4) \right\rangle$ at node i. $(D, 4)$ is an exit node, hence we go to the outer loop LTS (Figure 3(b)) where the context is $(B, 4)$. Then, there is a transition from 4 on the label i to node 5. We retain the traversed loop context $(D, 4)$ in the context string.

4.2 Our generic dataflow analysis framework

If z refers to a set of dataflow facts, we define $z^{q.c}$ as a set of facts annotated with context string $q.c$ where q refers to the latent context and c is the active context.

We define $c \;\to^{x}\; \alpha(c, x)$ as the transition from one state to another where $\alpha(c, x)$ is the new context state which is arrived from state c over the transition x in the current automaton (without leaving the current LTS). If there is no valid transition from $c \to^{x}$, then $\alpha(c, x)$ goes to the *epsilon state.* The epsilon state is the start node of the LTS and we define the ϵ-context string as the string whose active context is the start node of that LTS.

Our generic probabilistic dataflow analysis algorithm now proceeds as follows. For all possible context strings, we look for a possible transition from the current state. If such a transition exists, the current context string $q.c$ makes a transition to the state $q.\alpha(c)$ where $\alpha(c)$ refers to the new state in the active LTS. We apply the transfer function (of the dataflow analysis) over the meet of all predecessors only with the same context string for all contexts.

There is no valid transition to a node from any of its predecessors if the profiler didn't encounter the node during its run. We merge information from all paths to that node and do so by taking a meet over all predecessors over the ϵ-context string. Epsilon-transition implies that we compute facts without any profiling information same as normal dataflow analysis.

Our Dataflow solution at a node x is formally defined as follows: $DFSoln(x) =$

- $\sum_{c \in C, \alpha(c,x) \neq \epsilon} [\Upsilon(\bigotimes_{y \in pred(x)} DFSoln^{q.c}(y))]^{q.\alpha(c,x)}$: if at least one transition possible from current state
- $[\Upsilon(\bigotimes_{z \in pred(x)} [\sum_{c \in C} DFSoln^{q.c}(y)]^{q.\epsilon})]^{q.\alpha(\epsilon,x)}$: if x is entry or if no transitions possible from the current state

where Υ is the transfer function, $\bigotimes$ is the meet operator, $DFSoln(x)$ refers to the solution of all dataflow facts. $DFSoln^c(x)$ means the set of dataflow facts associated with context string c and $DFSoln(x) = \sum_{c \in C} DFSoln^c(x)$. The size of the latent string q is bounded by the number of loops in the CFG.

Example: A node n whose predecessors are p_1 and p_2 might have a transition from one of the context strings, $q.c$ in p_1 but not in p_2. Then, the resultant solution at n with the context string $q.\alpha(c)$ will be the application of transfer function over a meet of existing facts with the facts generated at p_1 alone. However, there could be another context string $q2.c2$ associated with p_2 which has a transition on n. Then, we would have two separate sets of dataflow facts associated with two different paths (or context strings).

The probability of a dataflow fact at a node n is computed using the context string associated with that fact. So, in Table 1, for node g, the probability of the dataflow fact with the context string $\langle(A,3),(B,4),(C,5),(D,2)\rangle$ is $Pr(3_{LTSa}) * Pr(4_{LTSb}) * Pr(5_{LTSc}) * Pr(2_{LTSd})$.

5. RELATED WORK

Profile-guided Analysis

Ammons and Larus [1] proposed a novel technique towards analyzing and optimizing programs by identifying and duplicating hot paths, thus creating a hot path graph. Their algorithm uses the data-flow tracing – one of the two techniques proposed by Holley and Rosen [3] for solving Qualified Flow-Analysis Problems. They preferred dataflow tracing over Context-Tupling (the second technique for solving qualified flow-analysis problems by Holley and Rosen). We, however, use context-tupling as it is more amenable to probabilistic analysis, with each context essentially mapping into a probability value for the tupled dataflow fact.

Path-Sensitive Analysis

ESP [8] performs property simulation (path-sensitive symbolic execution) to verify a specified program property by tagging each dataflow fact with the path it was generated on. We use path-sensitive analysis *only* on the paths that were seen by a profiler.

Dillig et al. [10] also proposed algorithms towards path-sensitive analysis. In contrast to the above algorithm, we segregate dataflow facts to compute probabilities with which a dataflow fact might hold at a program point. Also, we do not propagate facts along all static paths, but only along the paths seen by the structural acyclic path-profiler.

Probabilistic Analysis

Ramalingam [2] formulated probabilistic dataflow analysis for finite bi-distributive subset problems using edge profiles to infer the probability with which each fact holds true.

Mehofer and Scholz [9] used two-edge profiles to improve the precision of the probabilistic dataflow analysis. As they utilized execution history for calculating the probabilities of dataflow facts, they obtained significantly better results.

Silva et al.[5] formulated a scalable one-level context and flow-sensitive probabilistic pointer analysis framework by using linear transfer functions efficiently coded as sparse matrices. They used edge profiles for probabilities computation.

To the best of our knowledge, ours is the first attempt at performing probabilistic dataflow analysis over path-profiles on structure graphs. As paths encode the history of execution much better than edge or two-edge profiles, the probabilities computed using path-profiles should be much more accurate.

6. FUTURE WORK

Having developed a very simple proof of concept [13], our primary goal right now is to build a robust implementation of our analysis on a stable compiler framework. This would allow us to compare the improvements in the solution over other algorithms that relied on edge/two-edge profiles. We would also like to implement a few speculative optimizations and study the effect of an improved probabilistic solution on the performance of programs. The sensitivity of probabilistic dataflow analyis solutions and speculative optimization algorithms on improved profiling information make an interesting case study.

Having accomplished our primary goal, we intend to extend our algorithm to use the k-Iteration path profiles [11]. The k-Iteration profiles, in contrast to acyclic paths, are capable of recording much longer paths spanning multiple iterations of a loop. Our solution should hence improve still further as we would then be able to capture the effect of inter-iteration dependencies in a loop as well.

7. REFERENCES

[1] Glenn Ammons, James R. Larus. Improving data-flow analysis with path profiles. In PLDI '98.

[2] G. Ramalingam. Data flow frequency analysis. In PLDI '96.

[3] L. H. Holley, B. K. Rosen. Qualified data flow problems. In POPL '80.

[4] T.Yasue, T.Suganuma, H.Komatsu, T.Nakatani. An Efficient Online Path Profiling Framework for Java Just-In-Time Compilers. In PACT '03.

[5] Jeff Da Silva , J. G. Steffan. A probabilistic pointer analysis for speculative optimizations. In ASPLOS '06.

[6] T.Ball, P.Mataga, M.Sagiv. Edge Profiling versus Path Profiling: The Showdown. In POPL '98.

[7] T. Ball and J. R. Larus. Efficient path profiling. In MICRO '96.

[8] Manuvir Das, S.Lerner, M.Seigle. ESP: path-sensitive program verification in polynomial time. In PLDI '02.

[9] E. Mehofer, B. Scholz. A Novel Probabilistic Data Flow Framework. In CC '01.

[10] Dillig, I., Dillig, T., and Aiken, A. Sound, complete and scalable path-sensitive analysis. In PLDI '08.

[11] Subhajit Roy, Y.N. Srikant. Profiling k-Iteration Paths: A Generalization of the Ball-Larus Profiling Algorithm. In CGO '09.

[12] Y. Wu, A. Adl-Tabatabai, D. Berson, J.Z. Fang, R. Gupta. US Patent 6848100 : Hierarchical Software Path Profiling, 2005.

[13] R. Arun. Probabilistic Dataflow Analysis using Path Profiles on Structure Graphs : M.E. Thesis, Computer Science & Automation, IISc, 2009.